HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Sami Torniainen

IMPROVING EFFECTIVENESS OF REGRESSION TESTING OF TELECOMMUNICATIONS SYSTEM SOFTWARE

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology in Espoo February 25th 2008.

Supervisor: Prof. Raimo Kantola

Instructor: M. Sc. Martin Makundi

HELSINKI UNIVERSITY OF TECHNOLOGY Abstract of the Master's Thesis

Author:	Sami Torniainen
Title:	Improving Effectiveness of Regression Testing of Telecommunications System Software
Date:	February 25th 2008
Pages:	vii + 57
Faculty:	Faculty of Electronics, Communications and Automation
Professorship:	S-38 Networking Technology
Supervisor:	Prof. Raimo Kantola
Instructor:	M. Sc. Martin Makundi

Regression testing is an expensive process used to validate new software versions. The cost of regression testing accumulates from the time and resources spent on running the tests. To improve the cost-effectiveness of regression testing, typically two fundamentally different approaches have been utilized: test selection techniques or test automation.

The goal of this thesis is to explore whether test selection techniques or test automation should be utilized in order to improve the cost-effectiveness of regression testing at the target company. The target company is a Finnish telecommunications systems developer whose main deliverable is the product family consisting of several IP/MPLS routers. These routers are deployed in access-networks in order to provide QoS related services, such as, Intranet, Extranet, VPNs and corporate voice services.

To achieve the goal, the present regression testing practices are explored first. After that, a test selection technique and test automation are utilized in a limited scope. Their impact to cost-effectiveness will be evaluated empirically and the recommended method is presented.

Keywords: regression testing, black-box testing , test automation, test selection technique

TEKNILLINEN KORKEAKOULU Diplomityön tiivistelmä

Tekijä:	Sami Torniainen
Työn nimi:	Regressiotestauksen tehokkuuden parantaminen tietoliikennejärjestelmissä
Päivämäärä:	25. helmikuuta 2008
Sivumäärä:	vii + 57
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta
Professuuri:	S-38 Tietoverkkotekniikka
Työn valvoja:	Prof. Raimo Kantola
Työn ohjaaja:	DI Martin Makundi

Regressiotestaus on osoittautunut kalliiksi tavaksi varmistaa uusien ohjelmistoversioiden toimivuus. Kustannukset johtuvat ajasta ja resursseista, jotka kuluvat testien suorittamiseen. Kahta erilaista lähestymistapaa on tyypillisesti sovellettu regressiotestauksen kustannustehokkuuden parantamiseen: testien valintatekniikka tai testiautomaatio.

Tämän diplomityön tavoitteena on tutkia, kumpaa kahdesta yllämainitusta lähestymistavasta kannattaisi hyödyntää, kun regressiotestauksen kustannustehokkuutta halutaan parantaa kohdeyrityksessä. Kohdeyritys on suuri suomalainen tietoliikennejärjestelmien kehittäjä, jonka päätuote tällä hetkellä on IP/MPLS pohjainen järjestelmä. Tämä järjestelmä sisältää useita eri kapasiteetein varustettuja IP/MPLS reitittimiä, joita käytetään pääsyverkoissa. Järjestelmän avulla voidaan tarjota käyttäjille palvelunlaatuun pohjautuvia palveluja.

Diplomityön tavoitetta lähestytään tutkimalla tämän hetkistä regressiotestausta kohdeyrityksessä. Tämän jälkeen hyödynnetään sekä testien valintatekniikkaa että testiautomaatiota rajoitetussa mittakaavassa. Näiden menetelmien toimivuutta arvioidaan empiirisesti kustannustehokkuutta vasten.

Avainsanat: regressiotestaus, mustalaatikkotestaus, testiautomaatio, testien valintatekniikka

ACKNOWLEDGEMENTS

This thesis was carried out in a Finnish company that develops telecommunications systems in Espoo, during 2007.

I'd like to thank senior test engineers Teemu Nevala and Joydev Jana to whom I am grateful for all the help and comments. Especially, I want to express my gratitude for my instructor Martin Makundi for his support and interest in the subject. I want also to thank Professor Raimo Kantola for supervising the thesis and for his encouraging comments during the process.

Finally, and most of all, I want to thank my partner Heini and my parents Arja and Eero for their unconditional support during the whole of my studies.

Espoo, 25th February 2008

Sami Torniainen

TABLE OF CONTENTS

ABREVIATIONS

API	Application Programming Interface
AT	Automated Testing
ATM	Asynchronous Transfer Mode
AX-4000	Adtech AX-4000
BIOS	Basic Input-Output System
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineering
IP	Internet Protocol
LAN	Local Area Network
MPLS	MultiProtocol Label Switching
QoS	Quality of Service
RE	Risk Exposure
RTST	Risk-based Test Selection Technique
Tcl	Tool Command Language
TSI	Test Scripts Interpreter
UDP	User Datagram Protocol
VPN	Virtual Private Network

1 INTRODUCTION

1.1 Background

When developing a software system, it is important that the required level of quality is achieved. Even small errors in the system can be disastrous and costly to correct after the product has already shipped. Therefore, testing is a important aspect in the product development project of software system. To find faults and problems in the product design as early as possible, testing is done in many phases. A large software system is usually divided into many subsystems, and a subsystem is further divided into smaller modules. Software testing can then be separated into four phases:

1. unit/module testing,
2. integration testing,
3. system testing and
4. acceptance testing.

As software development involves changes to the code as a result of errors, or new functionality being added, experience has shown that these modifications can cause previously working functionality to fail. To check software's integrity against this kind of unexpected faults, regression testing is utilized. Regression testing can be accomplished on each of the four above-mentioned testing phases, and is ideally performed every time code is modified or used in the new environment.

However, regression testing is an expensive process used to validate new software versions. The cost of regression testing accumulates from time and resources spent on running the tests. For example, it can take up to seven weeks to run the entire test suite developed for a certain part of a software consisting of 20 000 lines of code. It has been estimated that regression testing may account for almost one-half of the cost of the overall software maintenance. [Kho06]

To improve the cost-effectiveness of regression testing, typically two fundamentally different approaches have been utilized: test selection techniques or test automation.
In general, test selection techniques aim to reduce the number of tests to run based on code-evaluation (e.g. finding un-initialized variables). Many studies [Agr93, Fra03, Gra01, Har88, Leu89, Rot97] have been made related to test selection techniques. In most of the studies, new algorithms are developed aiming to explore the code and detect the risky areas of the program more effectively than before. In addition, one work [Che02] proposes a test selection technique that aims to prioritize test cases based on risk-analysis. This technique evaluates the risk of a test case by using knowledge of the existing errors and their costs.

On the other hand, automating manual testing actions can also reduce the effort required in running tests. Lately, this has become a popular method to improve the cost-effectiveness of regression testing as an increasing number of enterprises are exploring their ways to utilize test automation. Automating tests is extremely cost-effective if they are run many times. However, the dilemma of test automation is that the benefits are achieved only after tests have been executed tens or even hundreds of times. Therefore, utilizing test automation should be analyzed carefully before starting to implement scripts. [Few99]

In this thesis, the target company is a Finnish telecommunication systems developer. At the moment, the main deliverable is the product family consisting of several IP/MPLS (Internet Protocol / MultiProtocol Label Switching) routers with various transmission capacities. Potential customers are major broadband and mobile service providers all over the world. They can use our equipment mainly on their access-networks for providing QoS (Quality of Service) related services, such as, Intranet, Extranet, VPNs (Virtual Private Networks) and corporate voice services to their business customers.

To stay in business, various customer needs must be satisfied. This means that new features are developed and added continuously into routers' software. This also means that they must be tested comprehensively before shipping to customers. After shipping,

the testing responsibility of the added features is transferred onto regression tests to enable more new functionality to be taken under test with higher priority. As a result, the size of regression tests increases as well as the effort they require to run. But, they must be executed to ensure the products' viability. Because of that, the target company is putting more and more pressure on improving the cost-efficiency of regression testing to cope with the increasing testing effort and limited resources.

1.2 Goals and Objectives

The goal of this thesis is to explore whether test selection techniques or test automation should be utilized in order to improve the cost-effectiveness of regression testing at the target company. This goal can be further divided into three sub-objectives that are presented below:

- exploring the present regression testing practices,
- utilizing a test selection technique and test automation in limited scale and
- evaluating them empirically.

1.3 Scope

Regression testing is limited into covering the integration testing level at the target company. In practice, this means that the software under test is seen as black-box. The program code will not be explored in any way. This has influence on e.g. which test selection technique will be utilized. In the light of test automation, regression testing is already partially automated at the target company. After exploring the present regression testing practices, we use them as a starting point when utilizing the test selection technique and test automation in order to improve the cost effectiveness.

The words scripts, test scripts or automated test scripts are mentioned many times. In this thesis, all of these will have the same meaning: an executable program designed for testing purposes, which is implemented using a scripting language.

1.4 Outline of the Thesis

The rest of this thesis is organized as follows. Software testing issues will be described in Chapter 2, providing the primary information on testing levels and techniques. After that, a more detailed look at regression testing is presented in Chapter 3, focusing on the risk-based test selection technique. An overview of test automation is provided in Chapter 4. In addition to discussion of the benefits and pitfalls of test automation, the main scripting techniques and the definition of automated testing are presented.

The actual work done for this thesis begins in Chapter 5, when we explore the present regression testing practices of the target company. In Chapter 6, we implement a test selection technique and test automation in order to improve the cost-effectiveness of regression testing. Chapter 7 presents evaluation of how the cost-effectiveness changes. Conclusions and ideas for future research are presented in Chapter 8.

Finally, Appendix 1 contains an example of risk exposures that are created in Chapter 6. Appendix 2 and Appendix 3 include the implemented scripts, whose creation is also presented in Chapter 6.

2 SOFTWARE TESTING

Software testing is the process of executing a program or system with the intent of finding errors, as Myers defines. [Mye79]. In addition, software testing aims to help identifying completeness, security, and quality of a developed program.

In general, it is very important that the quality of a new system is high before it is released to customers. If the system does not function as it is expected there is a high risk that its manufacturer loses money and gains bad reputation. The error does not need to be great for the customer to become unsatisfied. Fixing an error after the product has already shipped, is extremely expensive. To prevent this from happening, testing should be done in parallel with product design and as an important part of the whole development project.

As there are a number of different general definitions related to software testing used later in this thesis, they need to be explained. This chapter aims to explain those definitions, including testing levels and testing techniques.

2.1 Testing Levels

A traditional view is that testing is done at the end of the project after the system has already been implemented. For example, in the classical waterfall model (Figure 1), testing is deployed between implementation and maintenance –phases. In practice, the problem of this model is that there is not enough time to perform comprehensive tests or fix the arising problems, when testing is left to the end of the project.

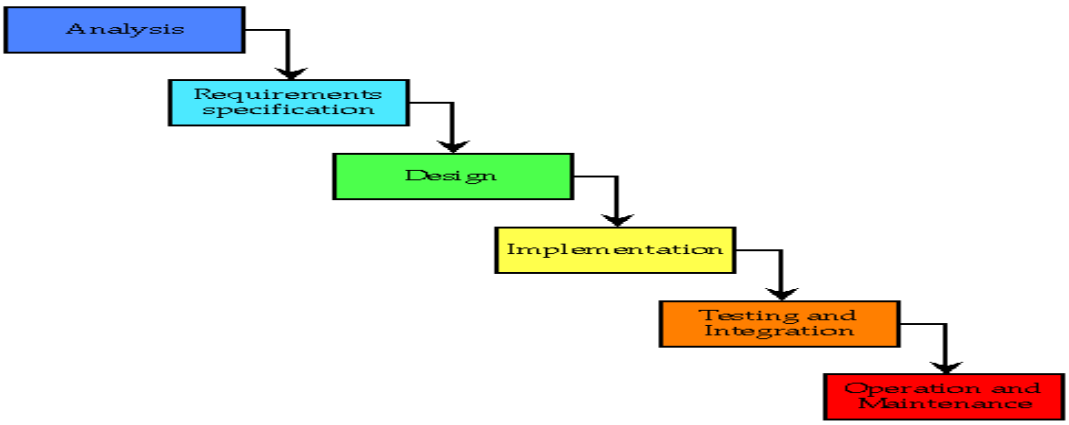

Figure 1. The waterfall model. [McC98]

The effect of finding a defect as early as possible is beneficial. It prevents the defect from being propagated onto higher testing levels, where it is more complicated and expensive to solve. For example, fixing a problem after the product has been built will have exponential cost in comparison to having fixed the errors earlier. In order to be cost-efficient, testing must be done in parallel with software development. [Few99]

The V-model (Figure 2) is one of the best ways to visualize the phases of testing. It illustrates when testing activities should take place. The four main process phases – requirements, functional specification, architectural design and detailed design – all have their corresponding testing phases. Implementation and functional design of modules is tested by unit/module testing, architectural design is tested by integration testing, functional specifications are tested by system testing and finally acceptance testing verifies the requirements. [Few99, Som96]

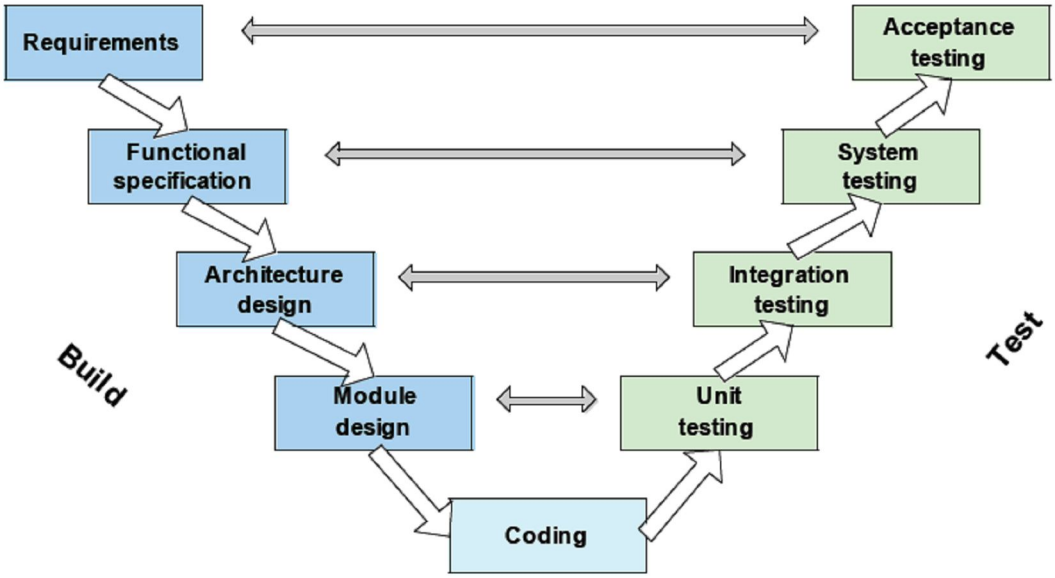

Figure 2. The V-model. [Itk07]

The V-model gets its name from the timing of the phases. Starting from the requirements, the system is developed one phase at a time until the lowest phase, the implementation phase, is finished. At this stage testing begins, starting from unit testing and moving up one test level at a time until the acceptance testing phase is completed. It is important to understand, that the different levels are tested simultaneously in multiple iterations. Adjustments are made to the product as defects are found. Only in this way can the V-model testing method become an effective and manageable testing process. This kind of level approach makes it easier to track the cause of an error and errors can be detected earlier. [Few99, Som96]

2.1.1 Unit/Module Testing

A unit is the smallest testable part of an application. Unit tests focus on functionality and reliability, and they typically consist of a series of stand-alone tests. Each test examines an individual component that is new or changed. A unit test is also called a module test, because it tests the individual units that comprise an application. Each unit test validates a single module that, based on the technical design documents, was built to perform a certain task with the expectation that it will behave in a specific way or produce specific results. [Ben]

2.1.2 Integration Testing

In integration testing, the individual components of the system are combined and tested as a group. Integration testing follows unit testing and precedes system testing. During it all the components and modules that are new, changed, affected by a change, or needed to form a complete system, are examined. Integration testing is specifically aimed at exposing the problems that arise from the combinations of modules, so-called module interface errors. [Som96]

Where system testing tries to exclude outside factors, integration testing involves multiple systems interfacing with each other, including those owned by an outside vendor, external partners, or the customer. Integration testing works best as an iterative process, when one unit at a time is integrated into a set of previously integrated components which have passed a set of integrations tests. [Som96]

2.1.3 System Testing

System testing is an investigatory testing phase that takes, as its input, components that have successfully passed integration testing. The goal is to ensure that the system now performs according to its requirements. System test evaluates both functional behavior and quality requirements such as reliability, usability, performance, and security, when all the modules are interacting together. This testing phase is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and inefficient memory usage. System testing may require involvement of other systems but this should be minimized as much as possible to reduce the risk of externally-induced problems. [Som96]

2.1.4 Acceptance Testing

Acceptance testing is the final stage in the testing process before the system is accepted for operational use. It gives confidence that the system has the required features that behave correctly. The system is tested with data supplied by the system producer rather than simulated test data. Acceptance testing may reveal errors and omissions in the original system requirements definition, because the real data exercises the system in ways different from the test data. Acceptance testing may also reveal problems in requirements where the system's facilities do not really meet the user's needs or the performance is unacceptable. [Som96, Mil01]

Other terms that can be heard in the context of testing include, e.g., regression testing. Regression testing is not a level of testing, but it is the action of retesting software and it is supposed to take place whenever changes are made to the code. It ensures that a new version of the software retains the capabilities of the old version and that no new defects are introduced due to the modifications. Regression testing can occur at any level of testing. For example, when unit tests are run, the unit may pass a number of such tests until one of the tests does reveal a defect. The unit should then be repaired and retested with all the old test cases in order to ensure that the changes have not affected the rest of

the functionality. Regression testing will be described with more detail in Chapter 3. [Bur03]

2.2 Testing Techniques

Software testing can be divided into static testing and dynamic testing. In static testing, the source code is analyzed before running it. Static analysis reveals, for example, syntax errors, unreachable code, undeclared variables etc. Reviews of all kinds of specification documents are also included in static testing. This type of testing is mostly used by the developer himself/herself, but not only manually. This process can also leverage automated tools, e.g., PMD or Clover for Java. Static testing is usually the first type of testing done on any system. [Kan99]

In dynamic testing, on the other hand, the code is executed and the response of a program to a set of input data is studied. This type of testing does not require knowledge of the implementation of software and there should be a test team to perform dynamic testing. Dynamic testing can be further divided into functional and structural parts, which are called black-box and white-box testing methods, respectively. These are discussed in more detail in the following sections. [Kan99]

2.2.1 Black-box Testing

In black-box testing, a system is viewed as a black box, as it is shown in Figure 3. It is an approach to testing where the tests are derived from specifications and architecture. The purpose of black-box testing is to find missing functions and defects from the system's interfaces, data structure, behavior, performance or initialization. The behavior of the system is then only be evaluated by studying its inputs and the related outputs. Black-box testing is also known as functional testing, because the tester is only concerned with the functionality and not the implementation. [Som01, Mye79]

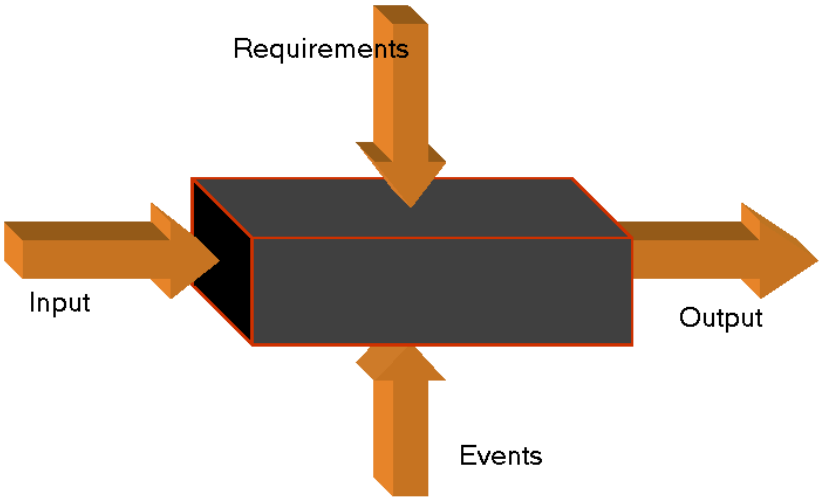

Figure 3. Black-box testing. [Dam00]

This method of test design is basically applicable to all levels of software testing: unit/module, integration, system, and acceptance. The higher the level, and hence the bigger and more complex the box, the more black-box testing is used as we are forced to simplify. In practice, black-box testing is hardly used in unit/module testing as the test design is then based on the code, not the specifications. In that case, white-box testing is usually used.

The key problem for the tester is to select inputs that have a high probability to induce anomalous behavior. In many cases, the selection of these inputs is based on the test engineers' previous experience.

2.2.2 White-box Testing

A complementary approach to black-box testing is white-box testing (sometimes called glass-box testing). As the name implies, the tester can analyze the code and use knowledge about structure of a component to derive the tests. White-box testing has many advantages. For example, the programmer can test the program in pieces and focus on one part at a time. The programmer can also find out which parts of the program have been tested and add tests that will cover the areas not yet examined. Unit and module testing are examples of white-box testing, because the tester, often the implementer of

the software module him/herself, uses his/her knowledge of how the module is implemented. White-box test cases should test different paths, decision points, execute loops, internal data structures, and check the state of the system at a given moment. [Som96, Kan99]

In this chapter, general things about software testing were presented, including testing levels and testing techniques. As the regression testing was mentioned only briefly, the next chapter presents it in more detail.

3 REGRESSION TESTING

IEEE (Institute of Electrical and Electronics Engineers) defines regression testing as follows: *"Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements."* [IEEE83]

During software development projects, there are several design changes that typically result from added new features and error correction work. Customers want new features in the latest releases, but still expect the older features to remain in place. This is where regression testing plays a role. In order to prevent quality from degrading, the new versions of software are retested using a combination of existing test cases. To accomplish this time consuming task effectively, test selection techniques and test automation are recommended. [Bur03]

This chapter aims to describe the most commonly utilized actions included in the regression testing process. Test selection techniques are presented because they can be utilized when the goal is to improve the cost-effectiveness of regression testing.

3.1 Regression Testing Process

The main phases of regression testing are usually all the same, and do not depend on the level of testing. First, feasible test cases are identified or created as necessary. When all needed tests are available, they are gathered to form a set. After this, the test set is executed against the target. [Hol04]

Holopainen [Hol04] has identified eight regression testing stages for software components. It begins from scratch, when there are no test cases for the component. After that, test cases are developed and testing continues until no defects are detected. In this model, K represents a tested component and K' represents its new or changed version. Respectively, T represents the test cases corresponding to the component K and

T’ represents new or modified test cases for the component K’. The states are then described as follows:

1. When the component K is tested for the first time, corresponding test cases T are developed.
2. The component K is tested against the test cases T. If defects are found, they must be verified and fixed (go to state 3). If no defects are detected, the testing ends (go to state 7).
3. Defects are verified and fixed. After that, a changed version K’ of the component K is developed.
4. Regression test cases T’ are selected from the original test cases T. Also, new test cases can be created to cover the fixed functionality of the component. These new test cases are added to both T and T’.
5. The component K’ is tested against the new regression test cases T’. If no defects are detected, testing ends (go to state 7).
6. Defects that are detected on regression testing are verified and fixed. After that, new test cases are selected or created for the fixed component (go back to phase 4).
7. Testing is finished.

Regression testing is also an expensive process. The cost accumulates from time and resources spent on running the tests. As mentioned earlier, it has been estimated that regression testing may account for almost one-half of the cost of software maintenance. In the next section, techniques for improving the cost-effectiveness of regression testing are presented.

3.2 Test Selection Techniques

To improve the cost-effectiveness of regression testing, various test selection techniques have been developed. During literature study, we found that the majority of the test selection techniques are based on exploration of the program code. For example, the firewall [Zhe05] approach tries to locate the harmful statements and functions whereas the dataflow [Gra01] and program slicing [Agr93] approaches are based on the recognition of harmful definition-use pairs. Due to the dependency on the code, these methods are applicable only in white-box-based testing. In light of this thesis, we see the software under test as a black box and the code is not meant to be explored. It is not even available for testers of the particular integration testing group.

Fortunately, one existing work presents the risk-based test selection technique that is developed especially for black-box-based testing. This method utilizes risk analysis to guide test selection. The risk-based test case selection technique is presented with more detail in the next section.

3.2.1 Risk-based Regression Test Selection Technique

Risk is a threat to the successful achievement of a project's goals. The tester's job is to reveal high-risk problems in software. Traditional testers have always used risk-based testing, but in an ad hoc fashion based on personal judgment. Chen et. al. [Che02] have developed a risk-based test selection technique that focuses on test cases that cover the risky areas of a program. In order to demonstrate how this method works, test prioritization based on risk exposure estimations is presented below.

First, the cost of fault $C(f)$ for each test case is estimated. Cost is categorized on a scale from one to five. There are two kinds of cost considered: the cost for the customer (e.g., losing market share and gaining bad reputation) and the cost for the vendor (e.g., high maintenance cost). The severity probability $P(f)$ for each test case is found by looking at the number of earlier defects and their severity. More accurately, this is accomplished by calculating the average sum of severities, multiplying that by the overall number of

defects and scaling the result into a zero-to-five scale. After that, the risk exposure (RE) is calculated for each test case according to the formula (1) [Che02]:

$$RE(f) = P(f) \times C(f) \quad (1)$$

Multiple scenarios covering one or more test cases are created. A scenario is a sequence of steps describing the interaction between the user and the system. These scenarios are effective since they involve many components working together. A traceability matrix is made, showing what test cases each scenario covers. The scenarios should cover the most critical test cases and as many as possible. The risk exposure for each scenario is calculated. The scenarios with the highest risk exposure are run first. After that, the traceability matrix is updated. New scenarios, with the focus on those test cases that have not yet been executed and with the highest risk exposure, are executed. [Che02]

Table 1 shows an example how risk exposure estimations are calculated for various test cases.

Table 1. Risk exposures of example test cases. [Che02]

Test Case	C (*t*) (1- 5)	Number of Defects (N)	Average Severity of Defects (S)	N × S	P (*t*) (1- 5)	RE (*t*) = C(*t*) * P (*t*)
t0010	5	1	2	2	2	**10**
t0020	5	1	3	3	2	**10**
t0030	3	1	1	1	1	**3**
t0040	3	1	4	4	3	**9**
t0050	3	2	3	6	4	**12**
t0060	3	3	3.5	10.5	5	**15**
t0070	3	0	0	0	0	**0**
…	…….	…	…	…	…	…

In this chapter, the regression testing process and test selection techniques were described. The main focus was on the risk-based test selection technique as it will be utilized later in this thesis. In addition to test selection techniques, another way to improve the cost-effectiveness of regression testing is automating tests. Fundamentals of test automation are presented in the next chapter.

4 TEST AUTOMATION

As regression testing mainly consists of re-execution of existing test cases numerous times, the task quickly becomes expensive to do manually. This is where test automation plays a role. Software testing needs to be effective in finding defects, but it should also be efficient in the way that the tests are performed as quickly and economically as possible.

Automating software testing can significantly increase the amount of tests that can be performed in limited time. Tests that would take hours to run manually can be automated to run in just minutes. Automated testing, however, comes with many pitfalls and limitations. For example, poorly organized automated tests do not contribute any value to the software development. To be successful, test automation demands careful analysis and design, which is often a considerable financial investment.

This chapter discusses the benefits and pitfalls of test automation. Scripting techniques are presented to illustrate how tests should be implemented. Automated testing aims to demonstrate what is needed to automate when reducing the testing effort to minimum.

4.1 Benefits of Test Automation

Test automation can enable some testing tasks to be performed far more efficiently than could ever be possible manually. There are also other benefits, including those listed below [Few99]:

1. Run existing (regression) tests on a new version of software. The benefit of automation is the ability to run regression tests quickly and easily in an environment, where many components of the software are frequently modified.

2. Run more tests to increase confidence. An obvious benefit of automation is the ability to run more tests is less time and therefore to make it possible to run them more often. This tends to produce a greater confidence that there will not be any unpleasant surprises when the product is released.
3. Better use of resources. Automating boring tasks, such as repeatedly entering the same inputs, gives a greater accuracy and frees skilled testers to spend more time in designing better tests. However, there always remains some testing that needs to be done manually. The testers can do the job far better if the majority of the tests are being run automatically.
4. Perform tests which would be difficult or impossible to do manually. Attempting to perform a full-scale live test of an online system with hundreds of users may be impossible, but the input from the same amount of users can be simulated using automated tests.
5. Earlier time-to-market. Once a set of tests has been automated, it can be run repeatedly far more quickly than it could be manually, so the testing elapsed time can be shortened. This can lead to faster project execution and earlier time-to-market.

4.2 Drawbacks of Test Automation

In addition to the benefits of test automation, there are also some drawbacks. Most test automation projects tend to fail due to unrealistic expectations, poor test practices, false assumptions or technical problems. A common misconception is the payback time of test automation: the benefits come only after test scripts are re-executed a number of times. This might mean after several consecutive version releases. [Few99]

A typical problem for a test engineer is to define what should be automated. The most common failure is trying to automate too much. It is not possible, or desirable, to automate all testing activities or tests. There will always be some testing that is much easier to do manually than automatically, or it is difficult to automate. The tests that should probably not be automated include tests that are run rarely, tests without

predictable results, tests where the results are easily verified by human, but are difficult to automate, tests where the software is very unstable, and usability tests.

Automating the tests too early is not a good idea either, because the specifications of the software tend to change several times during the product development project. In the worst case, the result is a lot of poorly automated tests which are difficult to maintain and vulnerable to software modifications. It is better to start small by identifying a few good, but diverse tests and automate them on an old and stable version of software. [Few99]

4.3 Scripting Techniques

The process of creating automated tests is called scripting. There are a number of different approaches that can be used in scripting. A common, but not very scalable, method is to record the manually performed tests and then replay this recording. The technique that the capture/replay tool uses for script generation is called linear scripting. [Few99]. It is a great way to learn how a new test tool works and to create quick scripts which are not intended for reuse. The most notable shortcoming of linear scripting is its fragility to change and unexpected events. Other approaches, such as structured and shared scripting, try to address the limitations of linear scripting.

In structured scripting [Few99], the scripts are programmed using common programming constructs such as sequences, selections (if, switch, case), iterations (for, while) and divisions into files (source and return). Error-handling code should also be included for robustness. These scripts become better documented and more structured, but the negative effect is that they can become too complex for a non-programmer to follow. In shared scripts [Few99], common functionality is captured into re-usable test components that can be included into test case specific scripts. The main reason for reuse is to eliminate duplication, thereby speeding up implementation of new tests and saving on maintenance costs. However, in order to be reusable, the shared scripts need to

be clearly documented, searchable and easily locatable. This requires clear guidelines for both naming and scripting itself.

The shortcomings of all the scripting techniques above are that test data is still hard-coded into the test script and therefore each test case requires at least one test script. Test data and its control functionality need to be separated in order to create more generic scripts. Therefore, a more powerful and viable methodology is the so-called data driven scripting technique. In this approach, complicated control scripts are created in order to simplify the actual test case data files. The aim is to produce a large number of test cases using only a small number of test control scripts. The automated test cases then read the test data from an external source, such as a file, rather than having the values hard coded into the scripts themselves. To accomplish this, the user must allow the scripts to take parameters. These parameters define the how the test functions, allowing the user to test a variety of scenarios with the same automated test scripts just by changing the test data parameters. [Few99]

4.4 Automated Testing

Automated testing is a testing process that does not require any manual involvement during execution. There are often some prerequisites that have to be in place before a test case can be run. Any processing that is carried out before the execution of a test or a set of tests is called pre-processing. For example, the tests may need some database with some specific customer records or a particular directory must contain some file with specific information. Post-processing, on the other hand, is any processing that is carried out after the execution of a test or a set of tests. An example of a post processing task is to clear up after a test case so that the next test case can be executed. [Few99]

If automated testing is wanted instead of just having some automated tests, the pre- and post-processing tasks surrounding the execution have also to be automated. We do not have automated testing if a tester must be involved during a series of tests to, e.g., restoring data. We cannot have unattended testing during nights and weekends if manual

involvement is needed. Figure 4 shows an example of tasks needed to perform a number of test cases and the difference between having automated tests and having automated testing. [Few99]

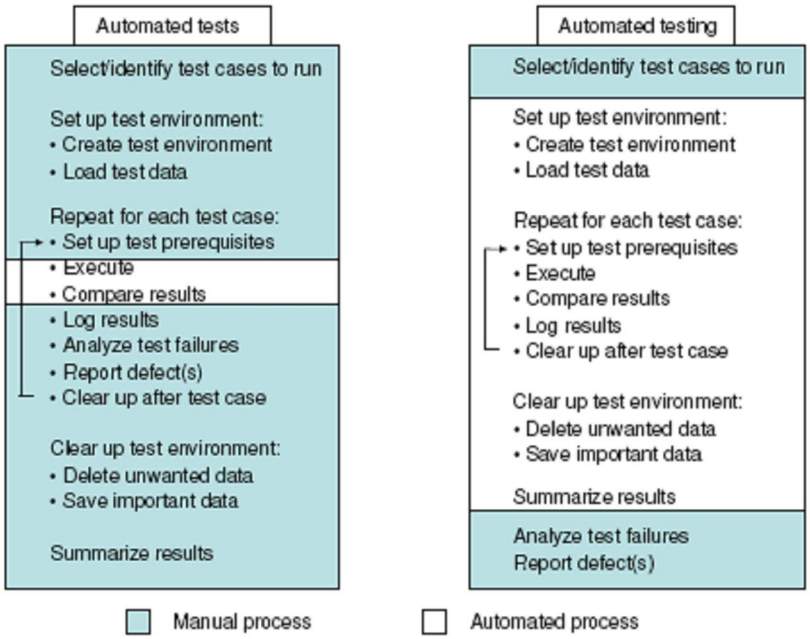

Figure 4. The difference between having automated tests and automated testing. [Few99]

The main difference between automated tests and automated testing is in the sequence of the tasks. With automated tests, the analysis of test results is done immediately after the comparison of actual and expected outcome. With automated testing, on the other hand, the analysis of the results is done after all the tests have run. For example, if a set of tests ran overnight then the tester will check the results in the morning and try to find out what has been wrong if some of the test cases have failed. [Few99]

Pre- and post-processing tasks can be implemented in scripts so that they are performed by the test execution tool. Many tasks are similar, and therefore shared scripts can often be used. Most pre- and post-processing tasks can also be implemented by using some form of command file (e.g. command procedure). [Few99]

In this chapter, general things related to test automation were presented, including benefits, drawbacks, scripting techniques and definition of automated testing. In the next chapter we explore the present regression testing practices of the target company, in which test automation is already partially utilized.

5 PRESENT REGRESSION TESTING PRACTICES

The purpose of this chapter is to explore the present regression testing practices of the target company. With the help of this, we can later design how test selection techniques and test automation can be utilized in order to improve the cost-effectiveness of regression testing. As mentioned in the Section 1.3, the scope is limited to cover the integration testing level. Therefore, the operations that represent the most appropriate actions of regression testing at the integration testing group are described here.

The actual work for this thesis begins here as we explore and document the regression testing process. This has not been done before at the target company. Overview of the test environment is presented to demonstrate how test equipment is utilized in big picture. Test scripts are described to give information about the scripting practices observed at the target company. Test tools and measurement devices are presented as they play a major supporting role in regression testing. They are discussed more when implementing the cost-efficiency improvements.

5.1 Regression Testing Process

Typically, in our particular case, the regression tests are data-circuit tests for verifying the creation and the deletion of supported circuit types, and the correct transmission of data on the links between the network elements and the measurement devices. Data circuits are configured and deleted using automated test suites, which are implemented as scripts. Test suites are equal to the pre- and post-processing activities of automated testing (presented in Section 4.3) as they are used to set up and clear up the nodes. They have been implemented earlier for integration testing purposes. Because of that, their creation is left outside of the regression testing process described here.

The first phase of regression testing is to select the test suites that are used to configure the target nodes. This is usually accomplished following ad-hoc principles, e.g., using previously gained knowledge of software changes. The set of available test equipment

also constrains which test suites can be executed at each time. This can be a bottleneck in busiest times when there is a lack of available test equipment resources. After selection is done, test execution begins.

Test execution consists of three phases: creating configurations, verifying the functionality under test and removing configurations. As mentioned earlier, we use test suites to create configurations to target nodes to activate the wanted functionality. This does not require more than clicking a mouse button few times in the text file editor as execution starts. However, checking that the functionality under test is working correctly is accomplished by generating and analyzing data transmission using measurement devices. This is performed manually. This phase takes most of the tester's work time in regression testing due to e.g. lengthy data transmission sessions for performance monitoring purposes. After the desired functionality is checked, nodes are cleared into initial state by removing previously created configurations. This is accomplished by continuing test execution, as the operations for removing the configurations are implemented at the end of the suite. Overall, there are tens of different test suites for configuring various test scenarios. Test execution is finished after they all are tested.

When test execution is finished, results are reported. There are normally two alternative outcomes: no failures detected or a fault is found in software. According to the outcome, alternative actions are taken. If there are no failures detected, testing can be stopped until a new software version comes out. If a fault is detected in software, we will report a defect into the bug tracking system. After that, a designer programs a fixed version of software. When the fixed software version is created, we run tests again.

The regression testing process described above is illustrated in Figure 5.

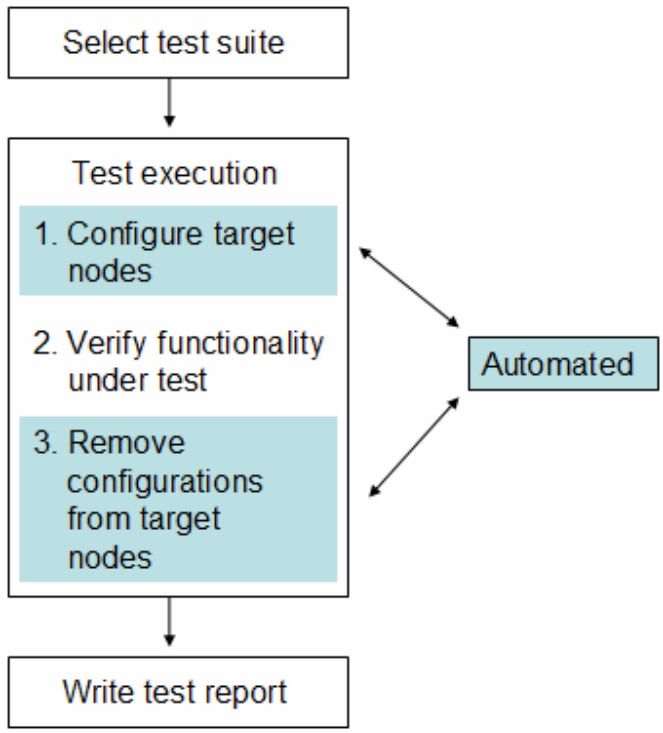

Figure 5. The regression testing process.

More closely, verifying the functionality under test is accomplished by generating test traffic and checking whether it traverses successfully through the target network. The target network consists of nodes on which the new version of software has been installed. A measurement device must be connected with e.g. Ethernet cables to both edges of the target network to enable data transmissions going through. Test traffic is set up according to the test scenario.

5.2 Test Environment

For conducting integration tests of new versions of routers' software an in-house build test automation environment TestNET is used. This is also utilized in regression testing. The test environment consists of measurement devices, different telecommunication network elements (test targets), communication servers and test scripts. [Kuo03]

TestNET is based on the client-server architecture. It is a multi-user system so it is possible to access the same network from many sources at the same time. Clients and servers are connected to the same LAN (Local Area Network) and they may be located completely separately from the test network. A client workstation is used to create, edit and execute test scripts consisting of commands for the test targets. Measurement devices interact with the test target according to the instructions given with GUI (Graphical User Interface), and carries out the given test operations. The simplified architecture of TestNET is illustrated in Figure 6. [Kuo03]

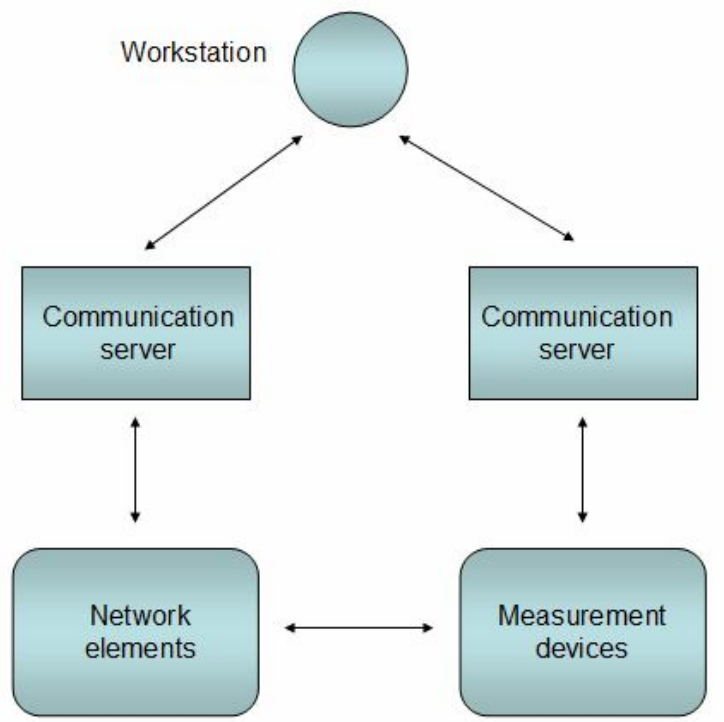

Figure 6. The TestNET's architecture.

5.3 Test Scripts

TestNET requires the test scripts to be written in Tcl (Tool Command Language). Tcl is an interpreted script language that has become increasingly popular among automated test systems, especially in the field of telecommunications and information technology. To organize and structure Tcl scripts in a logical way, the object oriented enhancement of Tcl, Itcl, has been used.

A test suite consists of the following Tcl scripts: a main file, a parameter file, test cases and procedures. When running a test suite, user executes the main file. A parameter file contains the actual data needed in testing, such as, IP-addresses, host names and serial numbers. To increase reusability and maintainability, a test case combines one or more procedures. Procedures are scripted according to principles of the shared and data-driven scripting techniques (presented in Section 4.3). In practice, this means that they are implemented in a form of so-called lover level functions that take parameters as input and return the actual test outcome as output. For example, removing node's serial number would represent a single procedure. All scripts are stored under version control. An example of a test suite is presented in Figure 7.

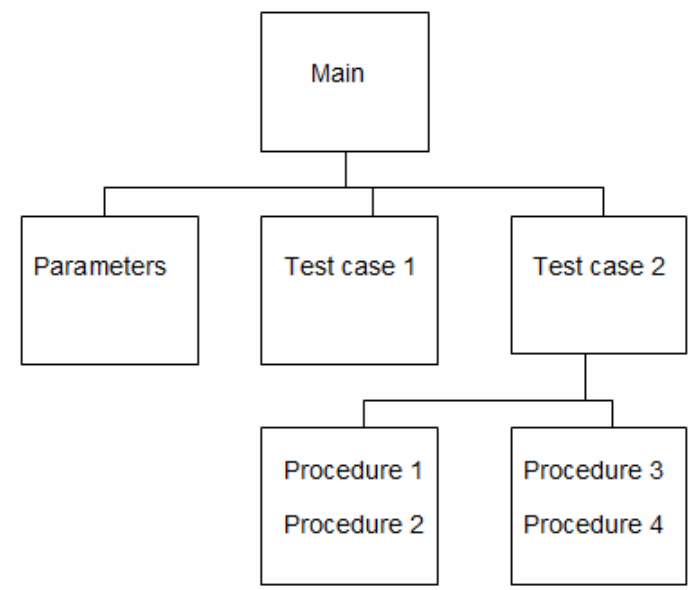

Figure 7. Test suite.

The integration testing group has put some development effort in creating reusable procedures to accomplish the commonly executed operations. For example, a system for storing test execution data has been developed. This system includes procedures for, e.g., initiating the creation of a HTML (Hyper Text Markup Language) file that is used as a test report, and printing the actual test outcomes and expected test outcomes into this file with their descriptions. This HTML-based test report is then checked after runs to determine whether all data circuits were created and deleted successfully.

5.4 Test Tools

For execution of the scripts, a Tcl interpreter called Test Script Interpreter (TSI) has been developed and added to TestNET. TSI is requested from TestBuilder that is a Codewright-based text file editor used in creation and edition of Tcl files.

Before TSI is ready to execute, it must read into memory all script files that are used in a test. This is accomplished by importing the script files at the beginning of the main file.
On user's request, TSI converts the script output into UDP (User Datagram Protocol) format and sends it out to the node's management interface through test lab's LAN. Tests are run in a TSI shell on a client workstation having a Windows operating system. By watching the shell, you are able to see how the test is going on. Multiple TSI shells can be open on the same workstation and therefore multiple tests can be executed simultaneously. [Kuo03]

For test management purposes, we use Hewlett Packard's Mercury Quality Center. This is a versatile test tool that contains large databases for storing test requirements, test case specifications and reported defects. All reported defects during the product's life time are stored here. Quality Center accomplishes, e.g., linking a defect with its respective test case. This kind of functionality eases the tester's effort in determining what tests need to be retested with a bug fix version. Quality Center is used via a web interface that is accessible from the company's intranet.

5.5 Measurement Devices

Measurement devices are used in generating and analyzing test traffic when we want to check whether the created data circuits are working correctly. In regression testing, Spirent-provided Adtech AX-4000 (Figure 8) is the most commonly utilised measurement device. AX-4000 is capable of using many different interfaces and supporting nearly all kind of protocols and services. It has a multi-port system that can test four different transmission technologies, including IP, ATM (Asynchronous Transfer Mode), and Ethernet and Frame Relay. [Cop04]

Figure 8. Adtech AX-4000.

The primary hardware components of AX-4000 are the chassis, a controller module and one or more interface cards. An interface card contains the circuitry to generate and analyze test traffic for a transmission medium. A controller module can manage multiple interface cards that can be mixed and matched to work with Ethernet, fiber etc. [Cop04]

The control of the device is primarily accomplished with the GUI application that is installed on windows workstations in the test lab. The GUI is relatively easy to use and the test setups can be saved on the hardware disk for later use. To support automated testing, the controller module includes Tcl/C API (Application Programming Interface). This enables remote control with Tcl and C-languages. In general, user needs to create a script that includes Tcl or C –based commands. These commands are processed in the

controller module, which parses them and performs the requested actions on interface cards. In this way, it is possible to automate test traffic generation and analysis to accomplish, e.g., unattended over-night test runs. The next section presents more information about programming the AX-4000 using Tcl. [Fly03]

5.5.1 Programming AX-4000 using Tcl

The article [Fly03] written by Flynt describes how AX-4000 can be programmed using Tcl. This article provides good background information, and focuses on implementing an Adtech script that is used for generating and analyzing Ethernet-based test traffic. According to Flynt, the general flow for an Adtech script can be presented as shown below:

1. load the Tcl extension,
2. initialize the connection to the AX-4000 controller,
3. lock an interface cards set,
4. create a generic interface object attached to the card set,
5. create an analyzer or generator attached to the interface,
6. configure the analyzer or generator,
7. run the test and
8. analyze results.

Next, the above-mentioned actions of an Adtech script are described in more detail.

For programming the device, the test environment must be prepared to accept the AX-4000 specific commands. For this purpose, a Spirent-supplied package is required. This package includes the Tcl extension and hardware BIOS files. In order to interpret the commands, the compiler must load the Tcl extension into memory. After that, hundreds of new commands become available, each of which has several subcommands. [Fly03]

However, before the commands of the Tcl extension have influence on the circuitry on AX-4000’s interface cards, we need to initialize the connection to the controller module.

This is accomplished by loading the hardware BIOS (Basic Input/Output System) files. This is required as interface cards contain FPGA (Field-Programmable Gate Array) components that cannot be configured unless the logic definitions of the BIOS files are loaded first. Initializing the connection can be done using a command of the Tcl extension and passing the location of the hardware BIOS files as an argument. [Fly03]

After the above-mentioned operations are done, we can program the FPGA components of AX-4000. To prevent tests from failing due to multiple users configuring the same interface card simultaneously, functionality for locking an interface needs to be created. [Fly03]

Operations related to e.g., creation and configuration of an interface card object, generator or analyser are used for building a test setup. In general, a test setup specifies what kind of test traffic is generated and how AX-4000 communicates with the nodes it is connected to. In regression testing, it is typical to create a lot of configurations for nodes to check performance under heavy load. As a result, test setups may be complex to create. Depending on the test scenario, we usually create tests setups that also include configuration of several network protocols and IP-addresses. [Fly03]

When a test setup is built, the test can be run by setting the generator transmitting test traffic through the target network towards the analyser. After transmission, test results are analyzed. This means actually fetching statistics from both the analyser and the generator, and comparing them. If they do not match, some traffic is lost and the test result can be set to fail. At least the result whether the test is pass or fail should be printed into the test report. [Fly03]

The connection between Tcl scripts and AX-4000 is illustrated in Figure 9.

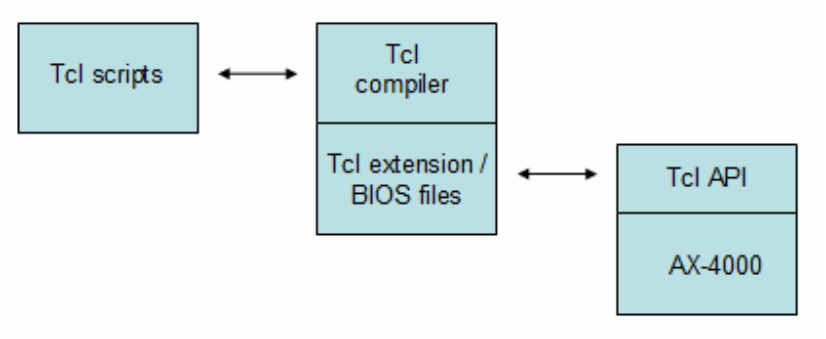

Figure 9. The connection between Tcl scripts and AX-4000.

In this chapter, the present regression testing practices were explored and presented. These practices are used as a starting point when we utilize a test selection technique and test automation in order to improve the cost-effectiveness of regression testing. That is presented in the next chapter.

6 IMPLEMENTATION

6.1 Goals

As we mentioned in the previous chapter, the greatest costs of regression testing arise from the effort that a tester consumes in executing the tests. More accurately, they are caused by the time consumed in generating and analyzing data transmissions for several different test scenarios, i.e., verifying the functionality under test. This takes time due to, e.g., lengthy data transmissions for performance monitoring purposes.

In this chapter, we will utilize a test selection technique and test automation to minimize the above-mentioned costs i.e. improve the cost-effectiveness. To accomplish this, we implement both methods in limited scope. Our scope includes ten different test scenarios. Each of them is configured active by running an existing test suite.

6.2 Design

The risk-based test selection technique (presented in Section 3.2.1) is chosen as the test selection technique to be utilized. This method does not explore the program code, which was the requirement as we see the program as a black-box. With this technique we aim to reduce the total number of tests run, but achieve better confidence on software quality than executing tests randomly. Our scope includes ten test suites that are prioritized using the risk-based test selection technique, i.e., risk analysis. Based on our prioritization, we will execute five of them on each test round. The number five is decided because we want to achieve considerable reduction in testing effort.

As a starting point for utilizing test automation, execution of tests is already partially automated: the wanted functionality is configured active and cleared from the nodes using automated test suites in each of the ten test scenarios. But, checking whether the software under test is working correctly is done manually by generating and analyzing test traffic using AX-4000. This provides a good platform for further development

towards automated testing. Automated testing (presented in Section 4.4) means that no manual intervention is needed during test execution. This is our goal. As AX-4000 includes a Tcl API, we can program it using Tcl to automate the actions performed formerly by a tester. As a result, the tester needs only to start the test execution from TestBuilder and check the results after the run to write a test report.

Figure 10 illustrates where the risk-based test selection technique and test automation have influence on the regression testing process.

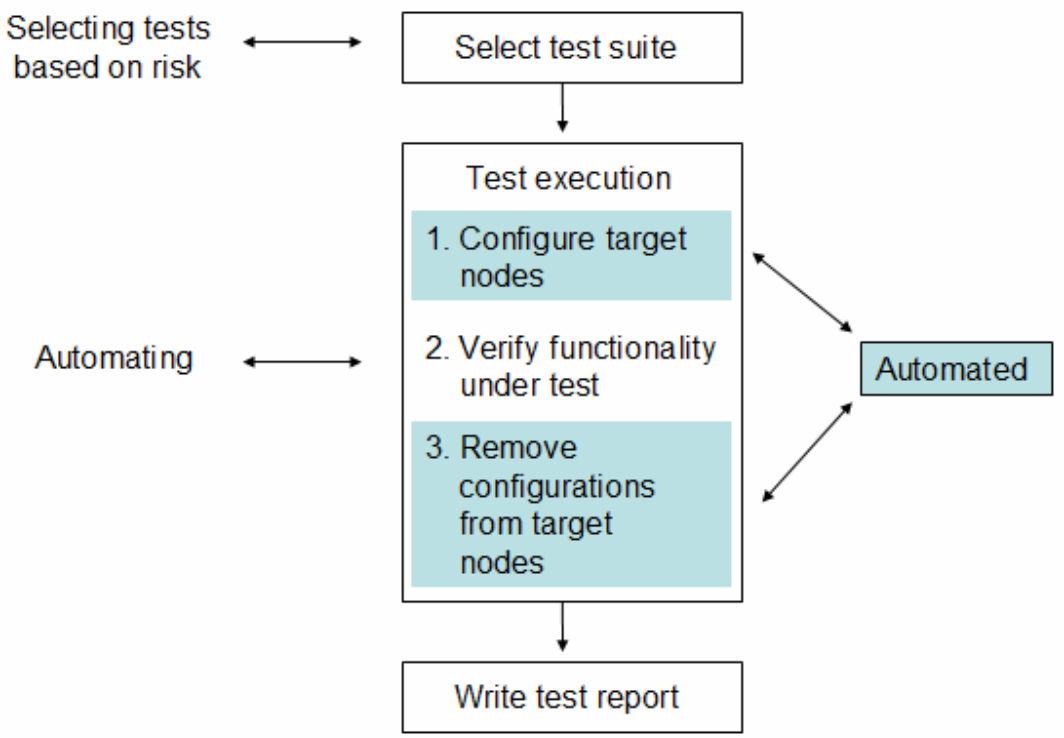

Figure 10. Utilizing the risk-based test selection technique and test automation.

During our work, we will keep track on the effort that is consumed in implementing both of these methods. This information will be used in the cost-effectiveness evaluation in the next chapter. The following sections describe in more detail how each of these methods were implemented.

6.3 Risk-based Test Selection Technique

In order to utilize the risk-based test selection technique, we first determined the attributes that are used for calculating risk exposures of test cases. As Chen et. al. presented, a risk exposure RE (f) of a test case depends on the severity probability P(f) and the cost C(f) as formula (2) illustrates:

$$RE(f) = P(f) \times C(f) = [\, N(f) \times S(f) \,] \times C(f) \quad (2)$$

To calculate severity probabilities, we checked how many errors were detected on each regression test case's coverage area, and what their severities were. This information was found from Quality Center that is the bug tracking application used at the target company. Quality Center's database contains all reported defects during the product's life time, and they are linked with the test cases that detected them. Severities of the defects were expressed with the following descriptions: low, medium or high. These were transformed into the scale from one to three, as we needed numerical values. For each test case, an average value of severities S(f) was calculated. After that, it was multiplied by the overall number of defects N(f) and scaled to the zero to five scale, where zero is low and five is high. For the test cases without any defects, P(f) is equal to zero. For the rest of the test cases, P(f) for the top 20 percentage of the test cases with the highest estimate N(f)×S(f) was five, and P (f) for the bottom 20 percentage of the test cases was one.

Costs C(f) are categorized on a one-to-five scale, where one is low and five is high. Two kinds of costs were taken into consideration. Customers can face costs, e.g., losing clients and gaining bad reputation. On the other hand, the vendor might face costs related to the resources consumed in fixing the errors. We determined the initial values for both of these cost types by consulting senior test engineers of the integration testing group. In principle, we put more weight on the core functionality than functionality that is rarely used. For example, the test case for configuring routing protocols was given higher cost than the test case for changing the node's host name. After both cost types were determined, they were summed up.

After the severities, the number of faults and the costs were determined for all test cases, we were able to calculate the risk exposures. At this point, some kind of database was needed for storing these attributes as they get updated every time a new software version is tested. Chen et. al. suggested that this database can be ideally integrated with the test tool used for test execution. However, we decided to store them into a Microsoft Excel spread sheet. In Excel, we also implemented a macro that performs the risk exposure calculation using formula (2).

In this way we calculated the initial risk exposures for the test cases. Summing up the test cases' risk exposures we further obtained risk exposures for test suites. The suite that gets the highest risk exposure will be executed first. This will be used as starting point in the next chapter when we start using this method to evaluate it. As an example, a spread sheet containing risk exposures of the IP regression test suite is presented in Appendix 1.

6.4 Automated Testing

To achieve automated testing, we created Tcl scripts that verify the functionality under test on behalf of a tester by generating and analyzing test traffic using AX-4000. Overall, scripts were created for ten different test scenarios and they were integrated with the test suites to be executed together. The design was based on Flynt's proposition of the general flow for an Adtech Tcl script. This was used as a guideline in identifying the actions required to implement here. In general, the best way to structure multiple actions into smaller functions turned out to be as shown below:

- locking interface cards
- preparing interface cards
- preparing test traffic
- run a test traffic
- analyzing results
- unlock interface cards

The main difference between the above-mentioned actions and Flynt's model is that we found it practical to implement also functionality for unlocking the reserved interface cards. This takes place after results are analyzed. We noticed that the interface cards locked through Tcl API can not be released using the GUI. The only option to unlock them turned out to be rebooting the whole chassis of AX-4000. However, this spoils the simultaneous data transmission initiated by other users, which was not a desired effect.

Before scripting, we stored the Spirent-supplied Tcl extension and the hardware BIOS files into test lab's shared network drive. The best way to implement scripts turned out to be observing the scripting practices of the target company, as it is essential to print test outcomes into a file. Using TestBuilder and the commands of Tcl extension, we created several procedures that carry out the above-mentioned actions in various test scenarios. The procedures were implemented to form of so called lover level functions to take parameters as input and return the actual test outcome as output. In order to store test execution data into a test report, we utilized the already created procedures for printing information into a HTML-file. With their help, we managed to print test outcomes and written descriptions of actions performed into a HTML-file. This file was used as a test report and is checked after run.

In general, the scripts will first reserve the wanted interface cards. Test setup creation includes preparing the interface cards for communicating with the target network and setting up the demanded test traffic. After that, transmission is initiated from the generator through a target network towards the analyzer. To analyze whether the test is a pass or a fail is accomplished by fetching statistics. We implemented this as a loop in which analyzer's statistics are compared against the generator's statistics. If they do not match with the packet loss criteria defined by the user, the test status is set as failed.

All procedures created here were grouped into separate test cases according to the test scenario they belonged to. The test cases were further integrated with the corresponding test suites. Integration was accomplished by scripting the following modifications into the main file of each test suite: (1) importing the files that contain Tcl scripts created

here and (2) running a test case that configures AX-4000 behalf of the tester. After integration, the test suites were successfully executed without manual intervention, which was our goal. The implementations of the procedures and the test case used in generating and analyzing IP test traffic are presented in Appendix 2 and Appendix 3, respectively.

In this chapter, we utilized the test selection technique and test automation that aim to reduce the testing effort. In the next chapter, we will evaluate how these methods work in practice.

7 EVALUATION

In this chapter we will empirically evaluate how the risk-based test selection technique and automated testing improve the cost-effectiveness of regression testing at the target company. In the following sections we will present test arrangements, goals, result analysis, recommendations and discussion.

7.1 Test Arrangements and Goals

The risk-based test selection technique (referred as RTST) and automated testing (referred as AT) are evaluated by utilizing them against real software versions created in the target company. In addition, we use the original testing practices as a benchmarking method in order to compare RTST and AT against it. Therefore, the benchmarking method is also run against the same software versions. Overall, five test rounds are accomplished with different software versions. Each version includes several bug fixes and added features. The test environment is TestNET and six target nodes are used with different scenarios.

With RTST, we decided to execute five of the ten test suites based on prioritization on each test round. The test suites that have the highest risk exposures are run. The figure five was selected as we wanted considerably large reduction in testing effort. Also, it is interesting to see how many bugs go undetected when the test coverage is 50 percentage of the overall test coverage. This gives a hint of how many bugs may be shipped to the customer. This has major influence on the costs in real life situation. With AT, and as well as the benchmarking method, all ten test suites are run on each test round.

To determine how each method influences on the cost-effectiveness of regression testing, we utilize the following measurements: (1) reduction of testing effort in time, (2) the number of detected errors and (3) the overall time consumed in deploying the methods. In a fact, the time consumed in deploying the methods was gathered during implementation phase in Chapter 6.

7.2 Result Analysis

Figure 11 shows the effort consumed in executing regression tests in this study. In the graph, the horizontal axis shows the software version under test, and the vertical axis shows the work hours consumed in executing the tests on the three different methods. These results show that RTST and AT reduced the testing effort when compared against the benchmarking method. AT required clearly the lowest amount of time of these methods – on average 1,6 hours per a software versions. RTST required significantly more time than AT – on average 7,1 hours per a software version. However, RTST was still a much less time consuming method than the benchmarking method, which required 13,2 hours on average per a software version. In the light of testing effort, AT is clearly more cost-effective than RTST.

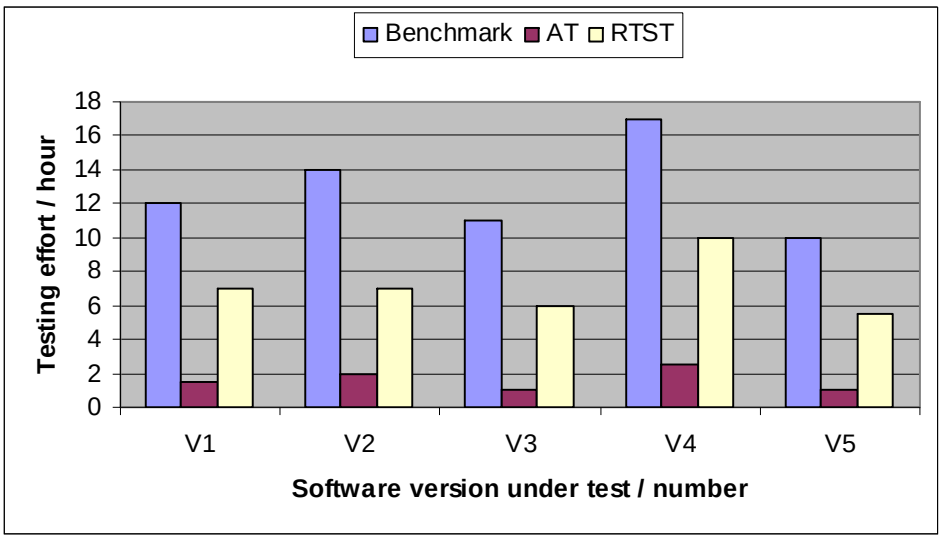

Figure 11. Testing effort.

The variation in the results between different software versions was mainly caused by the bugs that were detected. When a bug was detected, we analyzed and tried to reproduce it by re-running the detecting test suite. These actions generally increased testing effort with all three methods. With RTST, updating, calculating risk-exposures and prioritizing test suites required extra effort when testing the second software version, as some defects reported by other testers were not linked with the regression test cases in Quality Center. This cost some time as we linked them ourselves.

We also noticed that Tcl compiler (TSI) of TestNET jams occasionally during lengthy test execution sessions. This problem occurred only with AT when the tests were run unattended over nights. After jamming, re-running of test suites cost also extra time. This phenomenon did not occur with RTST as there were not such long continuous TSI sessions than with AT.

Figure 12 shows the number of errors detected with each method. In the graph, the horizontal axis shows again the version of the software under test, and the vertical axis shows new errors detected with each method. As the test coverage of AT and the benchmark method is the same, they did detect the same number of faults. As RTST was set to reduce the number of tests effectively, it covers only 50 percentage of the overall test coverage. Overall four bugs went undetected with RTST that were caught with AT. This occurred on software versions 2 and 5. Software version 5 did not contain any detected errors.

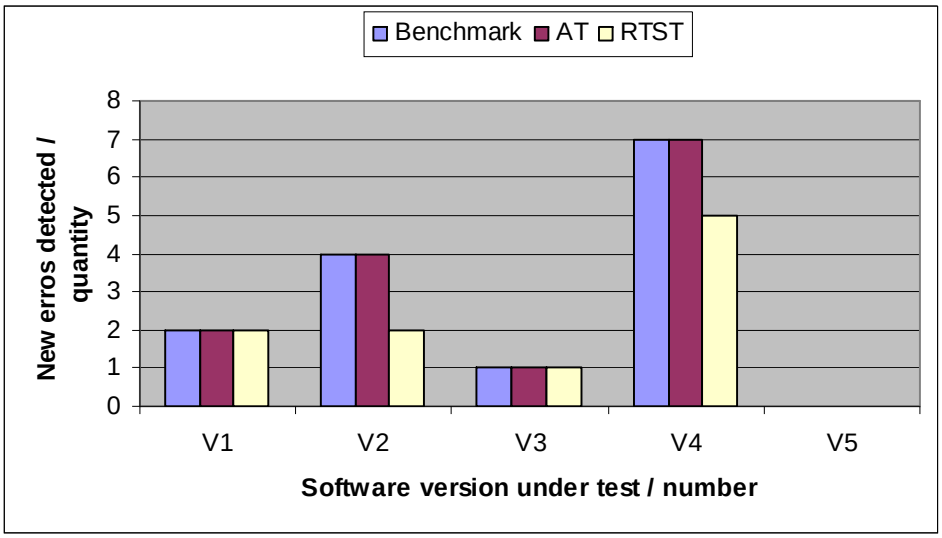

Figure 12. Detected errors.

The reason why RTST missed the errors was that the risk exposure calculations gave low priorities for the test suites that detected these four bugs on the other methods. Overall, RTST detected 71,4 percentages of the overall amount of detected errors in this study. This is quite good result because the test coverage was only 50 percentage of the overall coverage, and this shows that risk-analysis works. However, this also proves that it is likely that some faults will not be caught out if using only RTST. This should be included in the cost-effectiveness considerations as the errors shipped to customers may induce astronomical costs. In this light, AT would be a much more cost-effective method.

Figure 13 shows the effort consumed in deploying the RTST and AT methods. As one might expect, scripting the Tcl scripts to enable AT took much more effort than utilizing RTST. During the implementation phase, we kept track of the time that we consumed in implementing these methods. As a result, nearly 396 working hours were spent overall. Scripting of the Tcl scripts took almost 374 hours and this was the most time consuming phase of this work. Determining and calculating the initial risk exposures and storing them into a Microsoft Excel spreadsheet took 22 hours. These figures reveal that 94

percentage of the overall effort consumed in designing and implementing was spent with AT, and only six percent with RTST.

Based on these results, RTST is much more cost-effective to implement than AT. This did not require any great implementation costs such as e.g. producing scripts. Risk exposures were calculated quite easily as test cases and defects were linked together in Quality Center. This was found very practical in the light of RTST method. On the other hand, the scripting took quite much effort from us, because there were no Tcl code available related to programming of AX-4000. They were created from scratch.

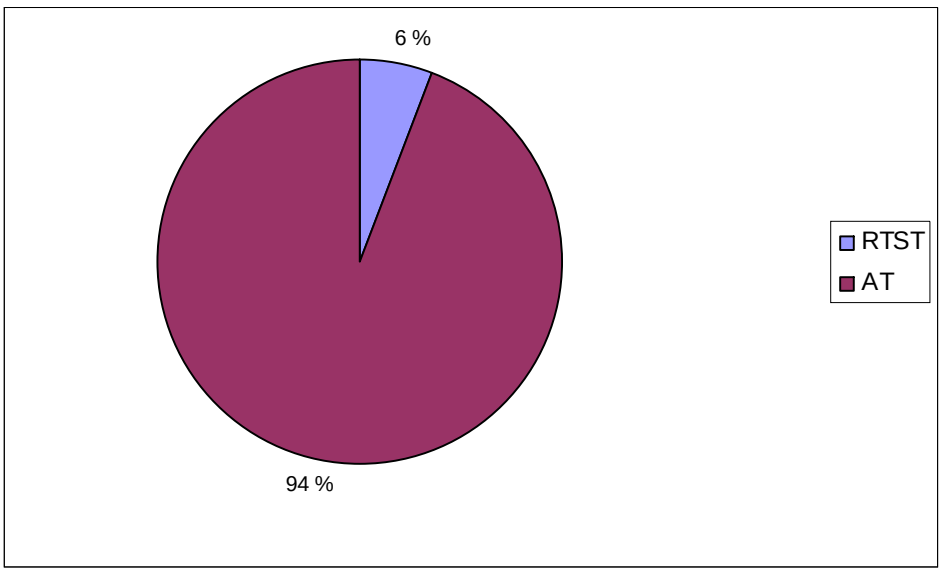

Figure 13. Deploying effort.

Despite the great design and implementation costs, we recommend that AT should be utilized in larger scale as a method for improving the cost-effectiveness of regression testing. It provided significant reduction in testing effort on a test round. RTST did also reduce the testing effort when compared with the benchmarking method, but it missed few bugs during the five test rounds. This might increase the total costs exponentially in

real life, which can not be tolerated even if the method's implementation did not require great effort.

To reduce the risks related to RTST, one could try to optimize the test coverage. As mentioned earlier, we had restricted it to cover only 50 percentage of the overall test coverage. For example, it could be explored how much the test coverage must be increased before we can reliably say that no errors will be missed. This would require hundreds of test runs to gain enough test data before we could present a reliable estimation. Therefore, it is not included in the scope of this thesis. However, this would be a potential area for future research. In the next section, we will discuss the costs and benefits of AT in more detail.

7.3 Costs and Benefits of Automated Testing

Using the data presented in the previous section, we can calculate how many test rounds it would require before AT pays itself back. On average, the time consumed in executing regression tests was reduced from 13,2 hours to 1,6 hours. Therefore, the savings per a test round were 11,6 hours. Design and implementation of the scripts required 374 hours to accomplish. If we assume that the working hours consumed in scripting and executing the tests have the same cost, the break-even can be calculated using formula (3):

$$E_i = \textit{effort invested to create scripts}$$
$$E_S = \textit{effort saved due to AT on a test round}$$

$$E_i = nE_s \qquad (3)$$

where n stands for the number of test rounds. The break-even can be achieved as follows:

$$n = \frac{E_i}{E_s} = \frac{374}{11{,}6} \approx 32{,}2 \quad \Rightarrow \quad 33$$

The break-even indicates that the tests must be run 33 times before the investment made in AT has paid itself back. This is not a large number if software is developed in small increments as it is done at the target company. New software versions are produced frequently for testing purposes. We believe that AT utilized in the scope of this thesis will pay itself back during the current project.

Utilizing more AT in the future would probably not require as much effort as in this thesis due to the knowledge gained during this work. The scripts that were created here can be used as guideline when more AX-4000 actions are automated. Also, some part of the Tcl code can be directly reused (e.g. the procedures for locking and releasing interface cards).

In the future, we should consider that scripting will be started at the beginning of the software development project. If this is accomplished, more AT will be utilized at the end of the project. Then, the regression tests can be executed far more quickly than at this moment. This would shorten the overall testing elapsed time, which can lead to faster project execution as programmers are dependent on the testing results. Furthermore, we can potentially achieve earlier time-to-market that would be a major advantage in competition against the other vendors. Then, AT would provide a great value addition to the target company.

8 CONCLUSIONS

The goal of this thesis was to explore whether test selection techniques or test automation should be utilized in order to improve the regression testing cost-effectiveness at the target company. The target company is a major Finnish telecommunication systems developer. The main deliverable is the product family consisting of several IP/MPLS routers with various transmission capacities. Potential customers are all major service providers in the world. They can use our equipment on their access-networks for providing QoS related services, such as, Intranet, Extranet, VPNs and corporate voice services to their business customers.

To achieve our goal, we started with exploring the present regression testing practices in the target company. The scope was limited to including the integration testing level. We identified that the test execution phase of regression testing causes the greatest costs at the moment. After that, we designed how test selection techniques and test automation can be utilized in order to reduce the execution costs i.e. improve the cost-effectiveness. They both were implemented in limited scale in this work.

As we see the software under test as a black-box, we selected the risk-based test selection technique as our test selection method. Using this method we can prioritize tests based on risk analysis. No code-evaluation is needed, which was the main influencing factor why this method was chosen. Quality Center was found a good information source when calculating the risk-exposures for test cases. Test cases were further used in prioritizing of test suites. Based on prioritizing, the five test suites that contained the highest risk exposures were executed on a test round. This reduced the number of tests to run to a half of the original size.

From the test automation point of view, testing was already partially automated. This provided a good starting point for further development towards automated testing. We realized that the measurement device mainly used in generating and analyzing data transmissions can be programmed using Tcl. We created Tcl scripts that control the device on behalf of a tester. As a result, the testing process became equal to automated testing, which means no manual intervention is needed during test execution. As a result, a tester needs only to start test execution from TestBuilder, check results after run and write a report.

When we had implemented test automation and the risk-based test selection technique, they were evaluated empirically. We regression tested five different software versions with both methods and analyzed the results. As a result, test automation provided significantly greater savings in testing effort than the risk-based test selection technique. The latter was less expensive to implement, but some bugs went ignored. In real life this might lead to a situation when a dangerous error propagates to a customer. This can not be tolerated as even small faults can cause great costs in gaining bad reputation. Based on the results, we recommend that test automation will be utilized as the method for improving the cost-effectiveness of regression testing in larger scale in the target company.

In the future, the scripting should be started at the beginning of the product development project. Then, the regression tests can be run far more quickly at the end of the project, which would shorten the overall testing elapsed time. This can lead to faster project execution and earlier time-to-market. Then, automated testing would provide a major advantage for the target company in competition against the other vendors.

8.1 Further Research

When creating the Tcl scripts for controlling AX-4000, we first tried to reuse the existing test setups made with GUI. This would have reduced the effort in scripting as test setups were the most complex parts to implement. However, we soon realized that this is not possible because there is not command for loading GUI-based test setup through Tcl API. We also asked Spirent whether they will create that kind of functionality in the future, but they had no such plans at the moment. Therefore, a useful issue for further research would be exploring whether it is possible to develop Tcl API for enabling the loading of GUI-based test setups in some way. This would reduce significantly the scripting effort.

With the risk-based test selection technique, we noticed that two or three test suites may get executed considerably more often than the others if using the method for a long time. The reason for this is that the number of faults probably becomes the dominant factor when calculating risk exposures. This means that a test suite that has detected a lot of faults in the past will also have high risk exposure in the future, even if it has not revealed any new defects for a while. To correct this behaviour, some kind of time window mechanism could be developed in order to define how long e.g. fault quantity information stays valid. If the risk-based test selection method will be utilized in the future, this is an issue that should be resolved.

One could also optimize the test coverage of the risk-based test selection technique to reduce the number of errors that are ignored. In practice, the test coverage can be increased until to a threshold in which the probability of missing a bug is as low as we want to. Exploring this threshold would be another interesting topic for future research. This would probably require hundreds of test runs to gain enough test data. Then, reliable estimations can be presented.

REFERENCES

[Agr93] Agrawal, H., Horgan, J., R., Krauser, E., W., London, S., A., "Incremental Regression Testing", Proceeding of the Conference on Software Maintenance", pages 343-357, IEEE Computer Society, 1993.

[Ben] Bentley, J., "Software Testing Fundamentals - Concepts, Roles, and Terminology", Wachovia Bank, Charlotte NC.

[Bur03] Burnstein, I., "Practical Software Testing: a Process-Oriented Approach", Springer-Verlag, page 176, 2003.

[Che02] Chen, Y., Probert, R., Sims, P., "Specification-based Regression Test Selection with Risk-analysis", IBM Press, 2002.

[Cop04] Coppens, J., "SCAMPI – A Scalable Monitoring Platform for the Internet", IMEC, 2004.

[Dam00] Damian, D., "Practical Software Engineering", lecture notes, Software Engineering, The University of Galgary, 2000.

[Few99] Fewster, M., Graham D., "Software Test Automation: Effective use of test execution tools", Addison-Wesley, pages 6-83, 1999.

[Fly03] Flynt, C., "The Tclsh Spot", The magazine of usenix and sage, volume 28, number 1, 2003.

[Fra03] Frankl, P., G., Rothermel, G., Sayre, K., Vokolos, F., I., "An Empirical Comparison of Two Safe Regression Test Selection Techniques", Proceedings of the 2003 International Symposium on Empirical Software Engineering, IEEE Computer Society, 2003.

[Gra01] Graves, T., Harrold, M., Kim, J., Porter, A., Rothermel, G., "An Empirical Study of Regression Test Selection Technoques", ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, pages 173-210, 2001.

[Har88] Hartmann, J., Robson, D. J., "Approaches to regression Testing", Proceedings of Conference on Software Maintenance, IEEE Computer Society Press, pages 368-372, 1988

[Hol04] Holopainen, J., "Regressiotestauksen tehostaminen", Pro Gradu, Tietojenkäsittelytieteen laitos, Kuopion Yliopisto, 2004.

[IEEE83] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 729-1983, IEEE Press, 1983.

[Itk07] Itkonen, J., T-76.3601 Basics of Software Engineering, Software Quality Practices -lecture slides, Software Business and Engineering Institute, Helsinki University of Technology, 2007.

[Kan99] Kaner, C., Falk, J. and Hguyen, H., "Testing Computer Software", 2nd edition, John Wiley & Sons, New York, page 124, 1999.

[Kho06] Khouru, M., "Cost-Effective Regression Testing", Seminar paper, Seminar on Software Testing, University of Helsinki, 2006.

[Kuo03] Kuosa, J., "Automated Test System for Broadband IP products", master thesis, Department of Electrical and Communications Engineering, Helsinki University of Technology, 2003.

[Leu89] Leung, HKN., White, L., "Insight into Regression Testing", Software Maintenance, Proceedings., Conference on, Volume 16, Issue 19, pages 60-69, 1989.

[McC98] McCormac, J., Conway D., "CSE2305 – Object-Oriented Software Engineering" –course, lecture notes, Monash University, 2005.

[Mil01] Miller, R., Collins C., "Acceptance Testing", RoleModel Software, Inc, 2001.

[Mye79] Myers, G., "The Art of Software Testing", John Wiley & Sonc, Inc., New York, 1979.

[Rot97] Rothermel, G., Harrold, MJ., "A Safe, Efficient Regression Test Selection Technique", ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, pages 173-210, 1997.

[Som96] Sommerville, I., "Software Engineering", 5th Edition, Addison-Wesley, pages 445-462, 1996.

[Som01] Sommerville, I., "Software Engineering", 6th Edition, Addison-Wesley, pages 442-444, 2001

[Zhe05] Zheng, J., "In regression testing selection when source code is not available", Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM Press, 2005.

APPENDIX 1: example risk exposures

Test Case	Cost C (1-5)	Number of defects N	Average Severity of Defects S	N x S	P (1-5)	RE = C x P
TC_GetUswInfo	1	0	0	0	0	0
TC_If_Configuration	4	1	3	3	2	8
TC_Ospf_Configuration	5	0	0	0	0	0
TC_Ping_1	1	0	0	0	0	0
TC_Ping_2	1	0	0	0	0	0
TC_Vpn_Configuration	5	4	2	8	3	15
TC_QosClassType	3	5	3	15	5	15
TC_Rsvp_Enable	2	0	0	0	0	0
TC_Ldp_Enable	2	0	0	0	0	0
TC_Ping_5	1	0	0	0	0	0
TC_Dhcp_Enable	1	0	0	0	0	0
TC_Dhcp_Ping	1	0	0	0	0	0
TC_Dhcp_disable	1	1	1	1	1	1
TC_Vpn_disable	5	1	2	2	2	10
TC_Rsvp_disable	2	0	0	0	0	0
TC_QosClassType_remove	3	2	3	6	3	9
TC_Ospf_disable	5	0	0	0	0	0
TC_If_remove	4	0	0	0	0	0
TC_Ldp_disable	2	0	0	0	0	0

APPENDIX 2: example procedures

```
proc RequestResources {args} {

	#set port user name according to workstation login or hostname
	set user $::env(USERNAME)
	if {$user==""} {
		set user [info hostname]
		if {$user==""} {
			set user "tcl_user"
		}
	}

	## loads BIOS files
	set biosdir "Z:/dxx.fan.verification/measeq/adtech/tcl-5-11/bios"
	set libdir "Z:/dxx.fan.verification/measeq/adtech/tcl-5-11/tclwin/libax4k.dll"

	foreach arg $args {

		if {[llength $arg]>1} {
			set device [lindex $arg 0]
		} else {
			set device $arg
		}

		switch $device {

			dlax {

				AX4000 dlax
				set lockList {}
				set ipAddressList {}
				foreach element [lrange $arg 1 end] {
					foreach {chassisNo cardNo devNo} $element {
						if {$chassisNo=="" || $cardNo=="" || $devNo==""} {
							error "AX4000 parameter missing!"
						}
						switch $chassisNo {
							1 { set ipAddress 172.19.13.142 }
							2 { set ipAddress 172.19.13.53 }
							3 { set ipAddress 172.19.13.54 }
							default {
							 error " Unknown AX4000 chassisNo: $chassisNo"}
						}
						lappend lockList $ipAddress $cardNo $devNo
						if {[lsearch $ipAddressList $ipAddress]==-1} {
							lappend ipAddressList $ipAddress
						}
					}
				}

				#initialise the requested chassises
				dlax init $ipAddressList $user $biosdir $libdir

				#lock the ports requested
				#if locking of one port causes error, unlock all ports and return with error
				set lockedPorts {}
				foreach {ipAddress cardNo devNo} $lockList {
					if {[catch {
						dlax lock $devNo $ipAddress $cardNo
					} errorinfo] != 0} {
						puts stderr " error in locking AX4000 port!"
						if {[llength $lockedPorts]>0} {
							puts stderr "error in unlocking AX4000 ports:"
							foreach devNo $lockedPorts {
								dlax do enet unlock $devNo
							}
						}
						#stop script execution here
						return -code error -errorinfo $errorinfo
					} else {
						lappend lockedPorts $devNo
					}
```

```
                                        }
                                } ;# end

                                default {error " Unknown device: $device"}
                        }
                }
}



public method prepareIpoEthTestDevices {devNo1 devNo2 vlanid1 vlanid2 args} {

        #optional parameters: default values
        set autoneg1 1
        set datarate1 MBS100
        set duplex1 full
        set autoneg2 1
        set datarate2 MBS100
        set duplex2 full
        ArgsToVars

        #disable use of VLAN capability if vlanid is negative number
        if {[expr $vlanid1]<0} { set enableVLAN1 0 } else { set enableVLAN1 1 }
        if {[expr $vlanid2]<0} { set enableVLAN2 0 } else { set enableVLAN2 1 }

        #find free names for objects (already created procedure)
        set if1 [FindFreeName Eth${devNo1}_]
        set gen [FindFreeName Eth${devNo1}Gen]
        set ana [FindFreeName Eth${devNo2}Ana]

        #set up one AX4000 interface card
        $this do interface create $if1 $devNo1 -ifmode IPoETHER -interface A
        $this do $if1 reset
        if {$autoneg1==1} {
                $this do $if1 set -mode NORMAL -autonegotiation $autoneg1 -enableVLAN $enableVLAN1
        } else {
                $this do $if1 set -mode NORMAL -autonegotiation $autoneg1 -datarate $datarate1 -duplex $duplex1 –enableVLAN
                $enableVLAN1
        }
        $this do $if1 run

        #check if we need to set up another AX4000 interface card
        #(if analyzer physically located in this port and not in the same port with generator)
        if {$devNo1==$devNo2} {
                set if2 $if1
                if {$enableVLAN2!=$enableVLAN1}
                        {error " VLAN must be enabled with both vlan parameters if only one port in use."}
        } else {
                set if2 [FindFreeName Eth${devNo2}_]
                $this do interface create $if2 $devNo2 -ifmode IPoETHER -interface A
                $this do $if2 reset
                if {$autoneg2==1} {
                        $this do $if2 set -mode NORMAL -autonegotiation $autoneg2 -enableVLAN $enableVLAN2
                } else {
                        $this do $if2 set -mode NORMAL -autonegotiation $autoneg2 -datarate $datarate2 -duplex
                        $duplex2 -enableVLAN $enableVLAN2
                }
                $this do $if2 run
        }

        #create generator and analyzer
        $this do generator create $gen $devNo1
        $this do analyzer create $ana $devNo2

        #bring generators and analyzers into their default state
        $this do $gen reset
        $this do $ana reset

        #generator trafsrc priority mode: round robin (may be required if many traffic sources in use)
        $this do $gen priority -mode rotate1

        #initialise ARP at the analyzer's port and clear reply lists
        set reqTimeout 1
        set reqRetries 3
        $this do arp terminate $devNo2
        $this do arp config $devNo2 -reqTimeout $reqTimeout -reqRetries $reqRetries
        $this do arp control $devNo2 clearList
```

```
		#if1 is for generator, if2 is for analyzer
		set devices [list $devNo1 $if1 $gen $devNo2 $if2 $ana]

		return $devices
}


public method prepareIpoEthTestTraffic {sourceMAC destMAC_A destMAC_B sourceIP destIP vlanid1 number_of_streams
                                        packetrate trafsrc devices args} {

		#optional parameters
		set packetsize 256
		set distribution Periodic
		set tos 0
		set up 0
		ArgsToVars

		if {[lsearch {Periodic RandSpaced} $distribution]<0} {
			error "AX4000::prepareIpoPosTestTraffic - invalid parameter $distribution"
		}

		#separate device names
		foreach {devNo1 if1 gen devNo2 if2 ana} $devices { break }

		### Generator:

		# Define the sequence and send it to the AX/4000 generator
		set seqNo $trafsrc
		set maxtaghandle [$this ipoEtherSeq2 $gen $seqNo $sourceMAC $destMAC_A $sourceIP $destIP $number_of_streams
				$vlanid1 $tos $up]
		# Packet distribution:
		$this do $gen dist $trafsrc $distribution -rate $packetrate
		# Datagram length = fixed
		$this do $gen dgramlen $trafsrc fixed -len $packetsize

		# Add every IP stream into ARP reply emulation, at the analyzer's port.
		# All IPs are mapped to the same MAC. Keeps already configured IP-MAC address pairs.
		$this addArpEntries $devNo2 $destIP $destMAC_B $number_of_streams


		### Analyzer:

		# Set up mAXTag subfilter (allows any ipv4 address)
		$this do $ana subfilter set maxtag


		# return test spec
		set test_spec [list $gen $ana $maxtaghandle $trafsrc]
		return $test_spec
}


public method doIpTest {test_spec testtime args} {

		#optional parameters
		set testInstance ""
		set pretesttime -1
		ArgsToVars

		set desc "AX4000::doIpTest2 - one-way data pass test (pre-test: $pretesttime secs, actual test: $testtime secs)"
		if {$testInstance == ""} {
			tsi::log print $desc
		} else {
			#Start test step
			$testInstance tStepStart -description $desc
		}

		#run pre-test (data failures allowed, the test wont fail)
		if {$pretesttime > 0} {
			if {$testInstance != ""} {
				$testInstance tStepTableInfo "PRE-TEST: (opening the pipes; errors allowed)"
			}
			puts "PRE-TEST: (opening the pipes; errors allowed)"
			$this startIpTest2 $test_spec
			SleepWithDots $pretesttime
			set result [eval $this stopIpTest2 [list $test_spec] -set_teststatus 0 $args]  ;#eval for args forwarding
		}

		#run actual test (zero tolerance)
		if {$testInstance != ""} {
```

```
                $testInstance tStepTableInfo -html "<b>ACTUAL TEST</b>:"
        }
        puts "ACTUAL TEST:"
        $this startIpTest2 $test_spec
        SleepWithDots $testtime
        set result [eval $this stopIpTest2 [list $test_spec] $args]  ;#eval for args forwarding

        if {$testInstance != ""} {
                #End test step
                $testInstance tStepEnd
        }

        return $result
}


public method startIpTest {test_spec} {

        #Separate list into variables
        foreach {gen ana maxtaghandle trafsrc} $test_spec { break }

        #Clear substreams and charting & histogram objects in analyzer
        $this do $ana clearList

        ### Start analyzer and generator
        $this do $ana run
        $this do $gen run
}

public method stopIpTest {test_spec args} {

        #Separate list into variables
        #In case multiple test specs are present, take device names from the first spec.
        foreach {gen ana maxtaghandle trafsrc} $test_spec { break }

        #optional parameters
        set testInstance ""
        set set_teststatus 1
        set direction1 {}
        ArgsToVars

        $this do $gen stop
        tsi::sleep 2
        $this do $ana stop
        $this do $ana refreshlist

        set testresult 0

        ### Check results


        #go through every traffic source given
        foreach {e1 e2 maxtaghandle trafsrc} $test_spec srcparams1 $direction1 {

                #exit from loop if test_spec list was not complete
                if {$trafsrc==""} {break}
                set res [$this checkIpTestResults $gen $ana $maxtaghandle $trafsrc -testInstance $testInstance $srcparams1]
                if {$res==1} {          set testresult 1 }
        }

        if {$set_teststatus == 1} {
                ReportTestStepResult $testInstance $testresult
        }

        return $testresult
}

proc FreeResources {args} {

        foreach arg $args {

                if {[llength $arg]>1} {
                        set device [lindex $arg 0]
                } else {
                        set device $arg
                }

                switch $device {
```

```
                                dlax {
                                        foreach element [lrange $arg 1 end] {
                                                foreach {chassisNo cardNo devNo} $element {
                                                        dlax do enet unlock $devNo
                                                }
                                        }
                                }

                        }

                        itcl::delete object $device
                }

}
```

APPENDIX 3: example test case

```
proc TC_IpDataTest {testSetName} {

   #Start test case
   set description "Adtect AX/4000 automated IP datatest"
   set testSpecificationDocument { "(see Quality Center)" }
   set testCaseName "ax4000"  ;#this should refer to the name of this file
   set testInstance [StartReportTestCase $description $testSpecificationDocument $testSetName $testCaseName]

            # parameters to configure Ethernet interface cards
            set adtech $::ADTECH
            set deVNo1 1
            set devNo2  2
            set vlanid1 -1
            set vlanid2 -1
            set autoneg1 1
            set duplex1 full
            set datarate1 MSB100
            set autoneg2 1
            set duplex2 full
            set datarate2 MSB100
            set args [list $autoneg1 $duplex1 $datarate1 $autoneg2 $duplex2 $datarate2]
            set devices [PrepareIpoEthTestDevices $testInstance $adtech $deVNo1 $devNo2 $vlanid1 $vlanid2]

            # parameters to configure IP traffic
            set sourceMAC 01.01.01.01.01.01
            set destMAC_A 00.b0.c7.06.a8.e0
            set destMAC_B 02.02.02.02.02.02
            set sourceIP 20.123.135.66
            set destIP  20.123.136.66
            set vlanid1 -1
            set number_of_streams 1
            set packetrate 225
            set trafsrc 1
            set test_spec [PrepareIpoEthTestTraffic $testInstance $adtech $sourceMAC $destMAC_A $destMAC_B $sourceIP $destIP $vlanid1
                                                    $number_of_streams$packetrate $trafsrc $devices]

            # Run IP traffic - duration 2700 sec
            set testtime 2700
            DoIpTest $testInstance $adtech $test_spec $testtime

   #End test case
   EndReportTestCase $testInstance
}
```